Plan-Driven Ubiquitous Computing

No Author Given

No Institute Given

Abstract. The goal of human-centered, ubiquitous computing should be to hide the details of the computing environment, allowing users to concentrate on their goals, rather than on the direct management of devices. This paper describes a system that operates at the level of goals and plans, rather than individual resources. It adaptively selects from its plan library that plan which is likely to best achieve the user's goal in light of his preferences and current resource availability. Once the plan and resources are selected, it monitors the execution of the plan, dispatching subtasks when they are ready to be executed.

1 Introduction

Recent work in ubiquitous computing ("ubicomp") research has produced resource discovery mechanisms [1, 2], programming environments [4, 9, 17], middleware [6, 10], and user interfaces [14, 16] that simplify the creation of user-centered applications that adapt to the different resources available in various computing environments. However, despite these advances, none of this research has addressed how a ubicomp application *should* use all the sensors, actuators, and other computing devices available in order to actually help a user carry out a task. Consequently, many ubicomp applications operate in a myopic mode that lacks the ability to proactively assist the user in his task, inform the user when he is unfamiliar with the task, and gracefully fail-over (and explain the rationale for this behavior) when resources become unavailable.

David Tennenhouse made a similar observation when reflecting on the advances in computing power that have made ubiquitous computing possible [19]. Due to the vast number of devices that are available for people to use and the speed with which these devices can operate, Tennenhouse proposes the need for new abstractions to better manage the behavior of these devices as they attempt to help us in our everyday lives.

Given this need for ubicomp applications to be smarter in the way they assist their users, we describe a plan-driven execution model for ubiquitous computing and describe a system implementation that supports such a model.

While earlier work in this area has proposed a task-driven model [3, 20] to ubiquitous computing, such approaches have treated a task as merely a set of applications, and any progress through the task is modeled through the application state. The fundamental problem with this approach is that it's focused on the wrong item; applications are only the tools used to carry out a task; they don't necessarily represent the task itself.

We choose instead to look beyond the task, and focus on the user's goals. For every task, the user has some goal in mind that would be accomplished by completing the task, and each step in the task (which may require a tool to complete) contributes to attaining that goal. We demonstrate how equipping ubicomp applications with an explicit plan-based representation of a user's task — a representation that includes the steps involved in the task, the resources needed by each task, and the causal and data flow relations between those steps — provides ubicomp applications a degree of *proactivity* that current applications do not possess.

In particular, plan-driven ubiquitous computing can provide the following benefits:

- A higher level of abstraction in designing ubicomp applications. From the perspective of application developers, working with a plan-representation of a task allows for a decoupling of application behavior from the actual implementation of that behavior. For example, plans can be used as an abstraction to aggregate resource requests.
- A higher level of abstraction in communicating intent. End users can present applications with the goal they want to achieve, instead of the individual actions that they will perform. The application could then produce the plan that would achieve that goal and then use this plan to guide its interaction with the user.
- Automatic configuration of environments. By knowing what the user plans to do, an application could reserve necessary resources and configure devices the user may soon need.
- Better informed systems. Given an explicit representation of a particular task, an application could guide users through the steps of that task if they are unfamiliar with it. If there are multiple things in the task that a user could tend to, such a system could also help direct the user's focus to the task that requires his attention most.
- Self-adaptive systems. If an application detects a component failure, it can use its representation of the user's plan to suggest an alternate course of action that would still achieve his intended goal.

All of our work is in situations within a variety of intelligent environments (IEs) ranging from kiosks to personal offices to meeting rooms. Although these differing physical settings have unique sets of equipment, they are driven by a common software base. This makes it particularly appealing to abstract away from the details of the specific devices, focusing instead on the common goals that one might bring into any of these facilities, the abstract plans that can achieve such goals, and the characteristics of the resources needed to populate each such plan. The system we describe in this paper allows for this to happen, and in doing so, we have applied techniques from the AI literature (namely, techniques in rational decision making) to the area of ubiquitous computing.

2 Scenarios

To illustrate the value of plan-directed execution, it is helpful to look at how an IE provides assistance in everyday activities. The scenarios we present are simple, but nevertheless common ones that show how our plan-directed execution model can naturally model and direct the IE's behavior.

2.1 Room Lighting

Howie's office can be lit through a variety of different mechanisms. He has an extremely bright halogen torchere in the corner, which can provide a warmer light than the fluorescent bulbs mounted in the ceiling. He also has a large window with movable drapes, which when open, let in a great deal of natural light from the courtyard outside.

When adjusting his lighting throughout the day, Howie can just issue simple commands like "brighten" or "darker", and the room can control the different devices to suit. Normally, he prefers to use the natural sunlight from outside, but the room recognizes that after sundown there isn't much point in adjusting the drapes, and instead will brighten or dim the torchere as necessary. During hot summer months, the room tries not to use the torchere at all, since the added heat just makes things uncomfortable. There are also times when Howie prefers to keep the drapes closed for privacy; his office can automatically adjust its lighting preferences based on those considerations.

2.2 Informal Meetings

Steve has just received a reminder that one of his undergraduate students has scheduled time today to discuss research progress. As the student enters, Steve issues a command to his office, so that the room gets properly configured for this session. Since the discussion is likely to consist of brainstorming new ideas, the room turns on microphones that can record the flow of discussion, and also enables the device that captures drawings and text written on the large whiteboard next to his desk. The room makes sure the environment's lighting is suitably bright, and then begins to record the meeting as it begins. When the meeting is over, the environment turns off the microphone and whiteboard devices, and stores the captured information for later review.

Steve occasionally sets up his room for other kinds of informal meetings. For example, when a student is going to be practicing a presentation, the room dims the lighting, turns on the projector, and uses both audio and video recording so that the student can review problem areas in the talk. For more private or personal discussions, the room ensures that no recording takes place while it closes the shades for privacy.

The scenarios above describe the interaction from the users' point of view, of course. However, this is meant to display the utility of the plan-based system. The users think only in terms of the goal state of the system ("darker", "set up the room for this meeting"), without having to delve into the sub-tasks necessary to achieve that state. By taking into account the current contextual situation, selecting the most appropriate series of steps to accomplish the goal, and executing them, the plan-based execution model simplifies the user's life considerably.

3 System Architecture

Our overall paradigm is that one makes a request for a goal to be achieved together with a set of preferences saying what properties are important to the requester (e.g. speed is more important than privacy). These preferences are then used to guide the selection of a plan capable of achieving the goal and to find an allocation of resources needed by the plan.

Generally, it is unreasonable to expect that an end user will be willing to explicitly state all of the preferences necessary to inform the system's decision making (although internal components of the system that make goal requests on their own behalf often can provide a complete set of preferences). However, there are two important sources of information that relieve the user of much of this burden: First, there are set of default preferences for each goal request, that are employed lacking other information. Second, context plays an extremely important role; in particular, task context provides very strong leverage in determining preferences. For example, the fact that a faculty member is meeting with a student, sets a preference for privacy; if the task is reading, then the preferred lighting level is moderate. We are accumulating a knowledge base of such task-context based preference rules; these are used to infer individual preferences which then are assembled into an overall preference statement. Thus, the overall paradigm is that context guides the selection of preferences and preferences guides the selection of plans and resources.

Figure 1 presents an architectural overview of the system that implements this paradigm. The system consists of four main components: a plan selector that chooses which plan best achieves a user's goal, a resource manager that assigns resources to these plans, and a plan executor and a plan monitor that carry out the plan. As the plan is carried out, the plan executor may consult the resource manager since conditions may change while the plan is running, thus requiring a new resource assignment to the plan

In the following sections, we describe our formal representation of a plan and examine in detail the pieces of our plan-based execution system.

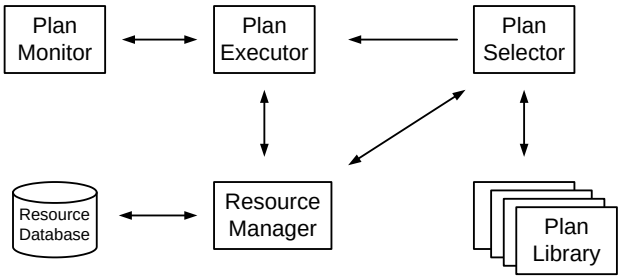

Fig. 1. In our system, a plan selector chooses the best plan to achieve the user's goal from a plan library. The resource manager then assigns resources to this plan, using resources from its resource database. The resourced plan is then passed to the plan executor to be carried out. The plan monitor informs the plan executor as progress is made through the plan

4 Plans As a Representation of User Tasks

A plan is a decomposition of a task into a set of partially-ordered steps. Each step has a set of pre-conditions that must be met before the step can be performed, and a set of post-conditions that hold after the step has been performed.

The partial ordering of the steps in a plan are defined by a set of data flow links, causal links and ordering constraints. A data flow link between step X and Y represents any material, whether physical or information, produced by X and used by Y. A causal link, represented as $X \xrightarrow{c} Y$, joins the post-condition of a step X with pre-condition c of another step Y, and can be interpreted as "step X achieves precondition c required by step Y". An ordering constraint between steps X and Y, denoted by $X \prec Y$, represents the fact that step X precedes (although not necessarily immediately) step Y; ordering constraints thus represent control flow in a plan and are used to prevent race conditions where performing one step would clobber either the effect of performing the other step or the pre-condition enabling the other step.

Each step in a plan may require a set of resources subject to certain constraints. For example, a plan step might require a computer and a display device, subject to the constraint that there is some mechanism available to pipe the computer's output to the display display (e.g. a video switch). Often there is more than one resource capable of meeting the plan's needs; for example, a remote screen network service is another resource that pipes a computer's output to a display device. In general, the choice of which resources to use is bound late in light of the actual operating conditions at execution time.

In addition, the plan may include hierarchical substructure. Substeps of the plan may either contain explicit internal plans, or they may be only the statement of a goal, requiring the selection of a plan capable of realizing that goal. As with resource allocation, this decision may be bound late, allowing current conditions to influence the choice.

Both the plan as a whole and the individual steps in the plan have a set of annotations. An annotation can be any information that is relevant to carrying out a plan, such as: which plan steps require special attention because they are more likely to fail; recommended orderings of steps, modulo the partial-ordering imposed by the plans causal and ordering links; instructions on how to use a service; and estimated service qualities provided by carrying out the plan (e.g., a plan may accomplish a task with a high degree of privacy, but in a very slow manner).

Plans can be represented graphically with a directed acyclic graph. A plan to prepare Steve's office for a student meeting is presented in Figure 2.

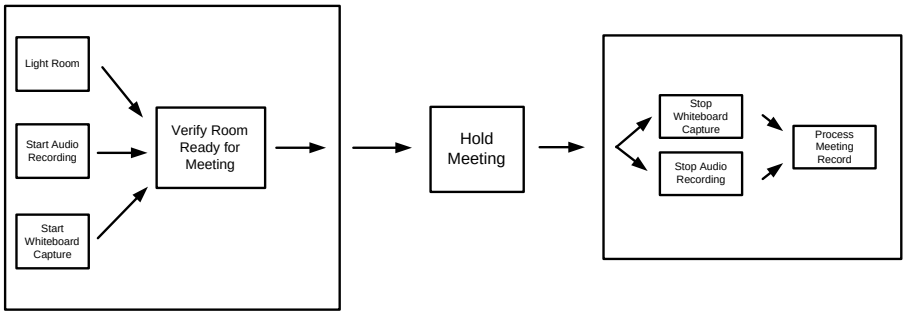

Fig. 2. A sample plan

In our system, we use Planlet [11], a middleware layer written in Java, to represent plans and user progress through plans. In addition to capturing the basic plan features described above, Planlet also allows plans to be defined hierarchically and for plans to be modified as they are being carried out. These two features allow plans to be reused by other plans and also allow for steps to be "late-bound" to a plan, if it is beneficial to defer the choice of a specific plan of action until execution time.

5 Plan Selection

One reason for adopting a plan directed viewpoint is to facilitate adaptation to a change in the conditions within which an intelligent environment operates. In particular, the set of resources available cannot be assumed to be constant: equipment breaks, new facilities are installed, a software system is migrated to a new physical facility, and other tasks may already be using a desired resource.

We deal with this by raising the level of abstraction at which users and other components of the system interact. Rather than centering on requests for a specific resource, or even for a specific type of resource, our interface centers on the requester's goals. Typical goals might be to present information or to

change the illumination level of a space. It is then the system's job to find a plan capable of satisfying the goal given the currently available resources.

We make the following observations:

- For any particular goal there may be many relevant plans: a room can be lit either by opening the drapes or by turning on a light; information can be presented on a printer, a display, or even verbally (using a voice synthesizer).
- Each plan requires a set of resources satisfying certain constraints. A plan to display information using the printer requires at least a printer, a connection to the printer, paper, and ink. A plan that displays information on a projector requires a projector and its display surface.
- These resources have a dynamically varying degree of availability: the printer may be busy, the projector's display surface may also be a white board that is being used for drawing.
- Plans that satisfy the same goal may deliver different qualities of service. Printing on a printer is certainly slower than using a display, but it usually is more readable.
- The qualities of service delivered by a plan may or may not be relevant to the user at the moment. The fact that a printer is slow doesn't matter if the purpose is to skim the information sometime in the future, while at other times, speed may matter crucially. Thus the user (or more generally the agent posting the goal) has a set of preferences over the different qualities of service: he might prefer speed to privacy on one request while on another his preference might be for image quality over everything else.

These observations have led us to a design that centers around cost-benefit tradeoffs; given all the possible ways to satisfy a goal, we should select that approach that best meets the user's preferences for quality of service and the minimizes the use of highly contested for resources.

We achieve this as follows (see Figure 3):

- First, we dynamically assign to each resource a "cost" reflecting its availability as well as the value of any consumable resources (e.g. paper, ink) used by the task.
- Secondly, we transform the user's service preferences into a numerical utility function that measures how well a particular plan satisfies the stated preferences.
- Third we maintain a library of plans capable of achieving each type of goal. Each plan is annotated with meta-data describing the types of resources required and the constraints on the overall ensemble of resources. Each plan is also annotated with meta-data that describes what qualities of service will be delivered given a set of resources. This meta-data is dynamic and procedural in character and represented as backward-chaining rules in a pattern-directed reasoning system.
- Finally, given a goal request and a set of preferences for qualities of service, the system considers each plan capable of achieving the goal. For each plan, it considers each set of resources capable of meeting the plan's resource

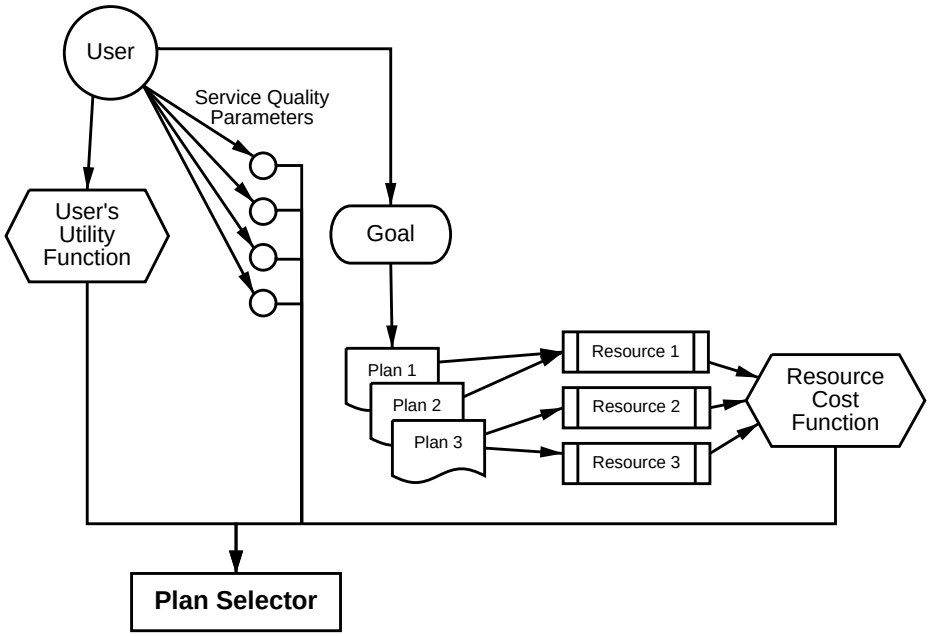

Fig. 3. The plan selection mechanism

constraints. Finally, for each plan and resource set it calculates the cost of those resources and the utility of the qualities of service delivered. It then chooses and executes that plan that maximizes the ratio of utility delivered to resources consumed.

One key step in all this is the process of transforming a set of user preferences over the service qualities into a single numerical utility function. Consider the (admittedly simple) goal of illuminating a room. The service qualities we include are: whether the lighting is natural or artificial, whether the lighting is bright, moderate or soft and whether privacy is maintained (since opening the shades impacts privacy). This is conveyed in a lisp-like notation:

```
(define-goal (illuminate)
  (privacy private public)
  (source natural artificial)
  (quality bright moderate soft))
```

The user's overall preferences are expressed as a collection of individual statements, each of which compares one set of features to another. For example, an individual quality preference statement might state that natural light and privacy is twice as good as bright artificial light. This would be written as:

```
(source natural privacy private
```

```
    (>> 2)
  quality bright source artificial)
```

Such a statement is taken as "all other things being equal." The user's overall preferences are then a set of such individual preferences. For example, the following would represent the fact that the user prefers natural light twice as much as artificial, moderate intensity three times as much as a bright intensity, and natural lighting five times as much as privacy:

```
'((source natural (>> 2) source artificial)
  (quality moderate (>> 3) quality bright)
  (source natural (>> 5) privacy private))
```

To convert a set of service quality preferences into a utility function we rely on techniques developed by McGeachie and Doyle [13]. The basic idea is to transform the assertions above into a graph; each node of the graph represents a particular assignment of values to the preference variables. Then an arc is drawn between any two nodes that are related by a preference statement. After all arcs are added, we then assign a value to each node by first assigning value 1 to any node with no outgoing arcs. For the other nodes, the value is found by iterating over all outgoing arcs and maximizing the product of the arc weight and the value of the target node of the arc. The utility function then simply looks up the node corresponding to a set of preference variables and returns that node's variable.

Attached to each plan is a set of meta-data saying what kinds of resources are required and how the quality of service parameters will be bound. In the example below, symbols beginning with a `?` are treated as logic-variables and are bound using unification (as in Prolog). This can be read as saying: "this plan provides natural light and no privacy. It requires a drapes-controller, and works only during the daytime, providing a lighting level equal to the amount of daylight."

```
(define-plan light-up-daylight
 :goal illuminate
 :features ((source natural) (privacy public)
            (quality ?amount-of-daylight))
 :resources (?drapes-controller)
 :resource-constraints
    ((the-type-of ?drapes-controller drapes-controller))
 :context ((time-of-day daytime)
           (daylighting ?amount-of-daylight)))
```

We have similar definitions for other plans that use the room's lamps and for plans that use both daylight and lamps. Figure 4 shows a trace of the reasoning process followed, given the above preferences and a request during a cloudy day (as determined by a call to a weather service):

```
:Lights Please :Privacy-Required (privacy-required [default Yes]) No

Method: LIGHT-UP-DAYLIGHT
Features: PRIVACY: PUBLIC, SOURCE: NATURAL, QUALITY: MODERATE
Resources DRAPES-CONTROLLER
Cost: 1 Utility 32 tradeoff 32.0

Method: LIGHT-UP-BOTH
Features: PRIVACY: PUBLIC, SOURCE: NATURAL, QUALITY: BRIGHT
Resources DRAPES-CONTROLLER,LIGHTS-CONTROLLER
Cost: 2 Utility 8 tradeoff 4.0

Best method: LIGHT-UP-DAYLIGHT,
   Features: PRIVACY: PUBLIC, SOURCE: NATURAL, QUALITY: MODERATE
Value: 32.0
Resources: DRAPES-CONTROLLER
```

Fig. 4. The reasoning process for finding a lighting plan

6 Resource Management

As suggested in the previous sections, the plan executor need not directly specify the precise devices that it will be using during the plan execution. Instead, we rely on a resource management capability, as described by Gajos et al. [8], which allows plans to specify their needs in terms of more abstract resource descriptions. Any abstract resource description made within the system (for example, requesting a "projector display"), is processed by the resource management system, which can find all possible projector displays, and perform arbitration to determine which of the existing displays is most appropriate. If the requester desires, it can also provide contextual information (such as "the room is currently in a 'meeting' mode"), which the resource manager can use to assist in arbitration.

This approach allows us to easily support the user's preferences (for example, if the user prefers a particular projector when he's in a room with multiple displays), and easily adjust to changing conditions at run time, without having to define separate plans for each individual resource that might be used. Because of this, plan designers can concentrate on defining the necessary steps that will accomplish a goal, and have the result usable in a wide variety of situations. Without it, designers would be forced to create a multitude of plans that enumerate the capabilities of just one environment or the desires of just one user.

The resource management framework used by our system allows the user to select a resource manager that suits his or her needs. These resource managers can range in complexity from simple name-based table lookup, all the way to high-level management tools that can perform sophisticated needs matching and re-allocate resources on the fly [7]. For the planning system, however, it makes

the most sense to have a simpler, lightweight approach which can incorporate the user's favored resources and contextual information without requiring a great deal of processing time, since some of the arbitration mechanism will be handled by the plan selector, and longer-term issues such as re-allocation of resources can be handled by diagnosis and recovery routines initiated by the plan monitor. For the planning system, the aim for resource management should be to quickly identify the most appropriate set of resources, and to respect the user's preferences during the selection process.

For this purpose, we've developed a new resource management module utilizing an in-house semantic network knowledge base [15]. This "Semantic Network Resource Module" (SNRM) allows the selection of favored resources, dependent on highly contextual situations, inside the environment's knowledge representation.

SNRM allows us to handle user preferences on a different dimension than quality of service. Just as there are times when a user will want to specify that he prefers "speed" to "quality", there are other times when he will want to say "when I'm in this location, I really want to use this display." Such considerations can be further biased with the plan selector's quality of service evaluation, but SNRM can fetch resources and make suggestions based on the user's contextual preferences. For example, users can describe situations like "In his office, Steve usually uses the first projector, except when he's having a meeting, when he uses the second" by setting up two different preference resources (see Figure 5), and returning the projector based on which preference resource matches the current situation most accurately.

```
(preferences-19834
  (requestor Steve)
  (request "projector")
  (result "projector-1")
  (conditions
    (Steve is-in office832)))
```

```
(preferences-28593
  (requestor Steve)
  (request "projector")
  (result "projector-2")
  (conditions
    (Steve is-in office832)
    (office832 mode meeting)))
```

Fig. 5. Two example SNRM preferences, described in a Lisp-like format for the sake of readability.

The conditions described for each preference are matched at request time to the current layout in the semantic network, allowing SNRM to tailor the resource list returned based on current data. For example, when the plan selector or the plan executor requests the abstract resource description "projector" on Steve's behalf, SNRM first finds all preferences in the network which have the given requester and request name. SNRM will then take the ensuing list and filter out any preference that doesn't match current conditions, finally ordering them based on the number of conditions that have matched. It can then return either

all possible results to the requester (for example, during the plan selection phase), or just the top-ranked result of the field.

The resource manager is used during plan selection (see section 5) to assist in determining the best plan for the user's goals. It is also used during the execution phase to determine the most appropriate resources for the current room state, since conditions in the room may change while the plan is running (for example, the user may have altered his preferred lighting setup, and will want the room to use the new preferences immediately).

7 Plan Execution and Monitoring

Once a specific plan has been chosen to execute, and resources have been assigned to the plan, the IE then carries out the plan. Plan-based applications need to provide two functionalities: a plan executor that actually carries out steps of the plan, and a plan monitor that watches over the progress of the plan.

The plan executor interprets the plan step and any annotations the step may have and then dispatches the necessary information to the software components that can complete the step. Steps are ready to be dispatched when all their preconditions are met and their ordering constraints are satisfied. In our implementation, steps that the IE can carry out consist of a generic resource description (e.g., a display), the name of a method offered by the service (e.g., "show") and a list of arguments to pass to that method (e.g., the name of a presentation). Our executor then calls that method of the specific resource that has been assigned to the plan.

While interpreting the plan and determining what steps are ready for dispatch, it is necessary to consider the fact that 1) a person, and not the application, may be the one who is performing a particular task (as noted in an annotation) and 2) a step may not complete successfully, regardless of who is responsible for performing that step. For these reasons, it is necessary to have a plan monitoring component.

If a user is the one who is performing a task, the plan monitor must be able to determine when he is finished with the task. There are a number of approaches to making this determination, such as using perceptual techniques to observe and infer that the user is done with a step, or adopting turn-taking or discourse modeling approaches to indicate when a user is done with a step. At the moment, we adopt a simpler approach to plan monitoring and listen for explicit user cues to indicate that he has completed a task, such as a button press or a vocal statement like, "I'm done."

To support monitoring for failure, plans have steps that test if the state of the world is within expected bounds. If this is not the case, then control is passed to a diagnosis and recovery unit to remedy the problem. Since the application already has a plan model of the task carried out by the user, model-based diagnosis is a natural approach to use to diagnose failures. Diagnosis and recovery is beyond the scope of our current work, but we discuss it further in Section 8.

8 Discussion and Future Work

We have described a system that guides an intelligent environment's behavior using a plan-driven execution model. Although the idea of supporting a user at different stages of his plan with computation is similar to the idea of traditional workflow, there are also a number of key differences. The main difference is that with workflow, the task structure is static, and it rigidly dictates what a person should do. Our plan-driven execution model, however, allows a much greater degree of dynamism at plan execution time by taking into account a user's context, expressed as his current preferences and by using a knowledge-based resource management system. The late-binding of steps to a plan and the use of the pre- and post-conditions of a plan for diagnosis and recovery are other features that are not provided by typical work-flow systems.

The issue of diagnosis and recovery is one area of future work that came out of our experiences with building this system. Currently, although our system can support any sort of plan step, including those that perform some sort of monitoring function, our plan monitor does not handle the case when these steps indicate that something has gone awry. Future implementations of the plan monitor will incorporate model-based diagnosis [18], a technique that takes advantage of our plan representation of tasks to determine a proper way of recovering from errors encountered during plan execution.

Improving the mechanism by which the plan monitor determines that a person has completed a step is another area of future work. In addition to the explicit cues currently provided by the user to indicate that he is done with a step, future implementations of the plan monitor will infer from perceptual cues, such as stereo cameras and activity maps [5], what activities are occurring and when they complete. This is work being done in conjunction with others in our research group, thus combining their bottom-up approach with our top-down methodology.

As noted in Section 5, the plan selection mechanism may be considering a number of plans, each of which may require a set of resources, and has to evaluate all of the possibilities with regard to the quality of service that is provided by each combination. Naturally, this raises the question as to whether the plan selection phase will require a great deal of time to reach its conclusion. For our current work, the number of plans has been fairly small, and the resource selection has been tailored to only those resources available in an environment, so this has not yet been a problem. However, it is possible to imagine more complex planning systems, and exploring resources in several environments at the same time. We are attempting to develop heuristics which will be able to quickly pare down the number of combinations that need to be evaluated, to keep this situation from becoming intractable.

When developing a more complex plan to achieve a certain goal, we have found it is often useful to design the plan in terms of sub-goals, and knit these into the more complex structure. However, it would probably be useful instead to simply specify the subgoal, and let the plan executor launch a new process to develop and select the sub-plan on the fly. The problem with this approach,

however, is that it could generate sub-optimal planning overall, since the sub-plan's resource requirements might impact the plan selector's utility function when creating the overall plan. We're examining ways of handling this gracefully, most likely by having the plan selector create simple bounded estimates of the sub-plan's utility, and using those estimates when generating the overall plan. Generating utility functions that can take advantage of these bounded estimates as they propagate up the plan's hierarchy might also be a useful approach.

9 Revisiting the Scenarios

We have seen how plans are selected, populated with resources and then enacted and monitored. We can now briefly return to our two motivating scenarios to see how these techniques contribute.

Lighting: "Howie can just issue simple command like brighten or darker". A request like "brighten" or "darker" is translated into a goal request with a set of preferences. Given the current state of the room's lighting, a command like brighten is taken to mean that all levels of illumination higher than the current are preferable to the current and lower levels. Default preferences specify that natural lighting is preferable to artificial and a contextualize default says that during the summer incandescent lighting is less desirable than other artificial lighting. These individual components of preference are then assembled and compiled into a utility function that is used to evaluate the various plans. As each plan is considered, the resource manager is consulted and returns specific combinations of lamps and drapes that are consistent with Howie's usual preferences. Finally, a plan that uses the overhead lights and the drapes is selected and then handed off to the executive for enactment. The DrapesManager and LightManager agents are called and the plan terminates successfully; had something failed, for example, had the drapesmanager not worked, then the diagnosis and recovery service would have been invoked to come up with a different plan.

Informal Meeting: "as the students enter, Steve issues a command ... so that the room gets properly configured for this session. Since the discussion is likely to consist of brainstorming.. the room turns on (information capture devices)... When the meeting is over the environment turns off the (capture devices) and stores the captured information.".. For .. personal discussions, the room ensures that no recording takes place while it closes the drapes..."

Here again we see how a context is used to infer a set of preferences. In particular, since the task is a meeting, it is important to use a plan that includes information capture (unless it is private meeting, in which case the preferences are reversed). These preference guide the plan selection, just as they did in the lighting example above. However, in this case the plan is more complex, involving four major sub-steps: setup the space, conduct the meeting, restore the space to its prior state, process and distribute the captured information. The plan monitor can easily determine when the setup phase is completed since it is responsible for conducting all the activities in that stage; but it must rely on human input to tell it that the meeting is over. The first two phases must

execute serially, but the last two phases (clean up and process the information) may execute in parallel since the plan has no constraints on their ordering and they do not contend for common resources.

10 Related Work

As mentioned in Section 1, although the high-level notion of a task has been introduced to ubiquitous computing, the formal notion of a plan is still missing. Wang [20] first introduces the notion of task-driven computing as a means of directing how the Aura infrastructure [9] should configure devices so that the user can maintain a sense of continuity as he moves from one place to another. In this model, tasks are modeled as a collection of services, and the service descriptions are used to find appropriate resources that provide those services. The emphasis in this view of task-driven computing is supporting a mobile user as he works on a general task that can be worked on with many applications (editing a paper or watching a video are the canonical examples). To this end, a service-oriented view of a task is sufficient.

However, this notion of a task is somewhat limited to office-work scenarios where providing the required services is all that could be done to assist a user. In other domains, tasks have a far richer connotation that includes a description of specific steps that must be done and dependencies between steps. In these domains, ubicomp applications have a far greater potential to assist users by actually performing the mundane plan steps or warning a user if he is doing something that would run counter to his task. Building ubicomp applications using a plan-driven execution model allows for added user benefit in these domains whereas a service-oriented view of tasks does not.

Christensen furthers the idea of a task-driven computing model by describing an early prototype of an activity-centered computing infrastructure to support hospital workers [3]. Similar to Aura, in this system, tasks are treated as first-class objects that serve as abstractions for applications, their state, and the data related to the activity. The most novel aspect of this system is a rule-based expert system that infers likely user activity based on context. Initial experiences with this system have been promising, but also raise many issues. Among these issues are how can the system provide additional value to its users; currently the system provides support for "follow-me" applications (mainly data access) but does not support non-composite activities. A plan-based model of activities would provide insight as to how additional support could be provided, and would make it easier to compose simpler activities into more complex ones.

Finally, in the Open Agent Architecture (OAA) [12], there is a facilitator agent that coordinates interaction between agents. In OAA, the facilitator agent performs task management by recursively decomposing a task into subgoals in a Prolog-like, backward-chaining manner. The facilitator then routes subgoals to those agents that can achieve them. While this is similar to our approach of resolving a goal by selecting from a set of plans (albeit at a lower level), it is unclear how the OAA facilitator dispatches tasks to agents. Since the primary

intent of OAA is to create a distributed agent architecture, it is unlikely that their dispatching mechanism considers user preferences when considering agent assignments.

11 Conclusion

In this paper we have shown how a ubiquitous computing infrastructure can be made highly adaptive through the use of reasonably standard AI planning and rational decision-making techniques. Our overall paradigm is goal and plan driven; the user provides a high-level request for a goal to be achieved together with a set of preferences saying what properties are important to the requester. The preferences are on the whole inferred from a knowledge based system informed by the task context and default settings. These preferences are then used to guide the selection of a plan capable of achieving the goal and to find and allocate the resources needed by the plan. Once a plan is selected that best meets the user's needs it is enacted by a plan executive that dispatches subtasks as they become enabled and that monitors their execution, checking that things proceed as intended. As described elsewhere [18], when breakdowns are encountered, model-based diagnosis can be used to characterize the failure and to help plan a recovery. The combination of explicit plan representations and late-binding, dynamic decision making allows the system to respond to variations in the execution environment in an adaptive manner.

References

1. W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, SC, December 1999.
2. Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison-Wesley, Reading, MA, 1999.
3. Henrik Bærbak Christensen and Jakob E. Bardram. Supporting human activities — exploring activity-centered computing. In *UbiComp*, pages 107–116, Göteberg, Sweden, 2002.
4. Michael Coen, Brenton Phillips, Nimrod Warshawsky, Luke Weisman, Stephen Peters, and Peter Finin. Meeting the computational needs of intelligent environments: The metaglue system. In Paddy Nixon, Gerard Lacey, and Simon Dobson, editors, *1st International Workshop on Managing Interactions in Smart Environments (MANSE'99)*, pages 201–212, Dublin, Ireland, December 1999. Springer-Verlag.
5. David Demirdjian, Konrad Tollmar, Kimberle Koile, Neal Checka, and Trevor Darrell. Activity maps for location-aware computing. In *IEEE Workshop on Applications of Computer Vision (WACV2002)*, Orlando, Florida, December 2002. IEEE.
6. Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16:97–166, 2001.
7. Krzysztof Gajos. Rascal - a resource manager for multi agent systems in smart spaces. In *Proceedings of CEEMAS 2001*, 2001.

8. Krzysztof Gajos, Luke Weisman, and Howard Shrobe. Design principles for resource management systems for intelligent spaces. In *Proceedings of the 2nd International Workshop on Self-Adaptive Software*, Budapest, Hungary, 2001.
9. David Garlan, Dan Siewiorek, Asim Smailagic, and Peter Steenkiste. Project Aura: Towards distraction-free pervasive computing. *IEEE Pervasive Computing*, 1(2):22–31, April-June 2002.
10. Jeffrey Hightower, Barry Brumitt, and Gaetano Borriello. The location stack: A layered model for location in ubiquitous computing. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems & Applications (WMCSA 2002)*, pages 22–28, Callicoon, NY, June 2002. IEEE Computer Society Press.
11. Miryung Kim, Gary Look, and João Sousa. Planlet: Supporting plan-based user assistance. Technical Report IRS-TR-03-005, Intel Research Lab, Seattle, 2003.
12. David L. Martin, Adam J. Cheyer, and Douglas B. Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1-2):91–128, January-March 1999.
13. Michael McGeachie and Jon Doyle. Efficient utility functions for ceteris paribus preferences. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 279–284, Edmonton, Alberta, Canada, July 2002. AAAI.
14. Brad Myers, Robert Malkin, Michael Bett, Alex Waibel, Ben Bostwick, Robert C. Miller, Jie Yang, Matthias Denecke, Edgar Seemann, Jie Zhu, Choon Hong Peck, Dave Kong, Jeffrey Nichols, and Bill Scherlis. Flexi-modal and multi-machine user interfaces. In *IEEE Fourth International Conference on Multimodal Interfaces*, pages 377–382, Pittsburgh, PA, October 2002.
15. Stephen Peters and Howie Shrobe. Using semantic networks for knowledge representation in an intelligent environment. In *PerCom '03: 1st Annual IEEE International Conference on Pervasive Computing and Communications*, Ft. Worth, TX, USA, March 2003. IEEE.
16. Shankar R. Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. ICrafter: A service framework for ubiquitous computing environments. In *Ubicomp*, Atlanta, Georgia, September 2001.
17. Manuel Román, Christopher K. Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, 1(4):74–83, October-December 2002.
18. Howard Shrobe. Computational vulnerability analysis for information survivability. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 919–926, Edmonton, Alberta, Canada, July 2002. AAAI.
19. David Tennenhouse. Proactive computing. *Communications of the ACM*, 43(5):43–50, May 2000.
20. Zhenyu Wang and David Garlan. Task-driven computing. Technical Report CMU-CS-00-154, Carnegie Mellon University, May 2000. http://reports-archive.adm.cs.cmu.edu/cs2000.html.